pyParticleEst – A Python Framework for Particle Based Estimation

Nordh, Jerker; Berntorp, Karl

2013

Document Version:
Publisher's PDF, also known as Version of record

ISSN 0280-5316
ISRN LUTFD2/TFRT--7628--SE

pyParticleEst – A Python Framework for Particle Based Estimation

Jerker Nordh
Karl Berntorp

Lund University
Department of Automatic Control
March 2013

Lund University
Department of Automatic Control
Box 118
SE-221 00 Lund Sweden

Document name
TECHNICAL REPORT

Date of issue
March 2013

Document Number
ISRN LUTFD2/TFRT--7628--SE

Author(s)
Jerker Nordh
Karl Berntorp

Supervisor

Sponsoring organization

Title and subtitle
pyParticleEst – A Python Framework for Particle Based Estimation

Abstract

Keywords

Classification system and/ or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280-5316

ISBN

Language
English

Number of pages
1-13

Recipient's notes

Security classification

Contents

1. Introduction

This technical report describes a python based library implementing a framework for assisting in solving estimation problems using particle based methods, primarily particle filtering and particle smoothing. For a theoretical introduction to the methods see e.g. [Arulampalam *et al.*, 2002] and [Doucet *et al.*, 2000].

When implementing these methods for a problem there are a significant number of steps that have to be performed and implemented every time. The aim of this work is to separate these steps using an object-oriented programming style where the developer only has to create a class describing the characteristics of the particular problem. This approach should help streamline the development process and reduce the number of potential bugs. The drawback is that using a generic framework can introduce an overhead compared to a fully specialised implementation. Thus the main target for this library currently is for either problems where the major computational effort lies in the problem specifics, or as a help for prototyping before writing optimized problem specific code (if needed).

2. Overview

The library introduces a number of classes that describes certain collections of data and provides methods for common manipulations of them. For a diagram of all classes and their methods, see Figure 2 in Section 7.

- ***Particle***, the class that has to be reimplemented for each new problem
- **ParticleApproximation**, a collection of *Particle*s and their associated weights. The particles and weights constitute a sampled probability density function
- **ParticleTrajectory**, a list of **ParticleApproximation**s ordered so that each index represents a successive time-step for the filtered estimation problem
- **SmoothTrajectory**, create a smoothed estimate from a filtered trajectory
- **SmoothTrajectoryRB**, create a rao-blackwellized smoothed estimate

The typical work flow for solving a filtering estimation problem is:

- Create a class describing the mathematical model for which the estimate is to be made. It has to implement three methods describing the noise affecting the input, how the system state evolves over time, and lastly a method to evaluate how likely a measurement is.
- Instantiate a number of particles representing the prior belief of state, use these to create a ParticleApproximation object.
- Create a ParticleTrajectory that is initialised with the ParticleApproximation object
- Feed the ParticleTrajectory object with all inputs and measurements

The particle trajectory now contains a sequence of ParticleApproximations where each index represents the posterior probability conditioned on all inputs and measurements up to that time.

3. Classes

3.1 ParticleApproximation

A **ParticleApproximation** object contains a NumPy array holding all the particle objects. It also contains a second array keeping track of the weight for each particle.

It provides methods for resampling the distribution so that all particles have equal weights, changing the number of particles and sampling a single particle. It is also possible to the extract the N best particles, where N is an integer between 1 and the total number of particles.

Changing the number of particles is accomplished by cloning or discarding particles according to the weight distribution. This is useful if the estimation problem has the property that the number of particles needed for a good approximation changes over time or depends on the measurements or input. The application-specific code could then detect this and instruct the framework to change the number of particles depending on the situation, providing both fast execution time and more efficient memory usage.

3.2 ParticleTrajectory

A **ParticleTrajectory** object contains a list of particle approximations. The *update* method takes an input to the system and then appends a new approximation to the end of the list with the updated estimate. It also provides a *measure* method which processes a new measurement, and if desired automatically resamples the underlying particle approximation if the weights of the particles are too unevenly distributed. This is detected by calculating the so called "number of effective particles" and comparing that to the total number of particles.

Additionally it implements some helper methods for calculating one-step forward probability densities which are required for the backward rejection sampling step if smoothing is used.

Methods are provided for creating a new particle trajectory object initialized by the estimation from the current one and for extracting the signal that have been used to calculate the estimate. This is useful for e.g running partially overlapping filtering/smoothing when it is not possible to do smoothing over the entire dataset.

3.3 ParticleFilter

This object type is typically used inside a **ParticleTrajectory** object where it provides the actual logic for creating the perturbed input signal and handling the resampling after new measurements are provided.

3.4 SmoothTrajectory

This object is created from a **ParticleTrajectory** and contains smoothed estimates where all the linear Gaussian states are collapsed to a single point estimate. These are stored in a list containing so called collapsed objects, it

is the responsibility of the problem specific class to define a class for this and providing a method, *sample_smooth*, for creating them.

3.5 SmoothTrajectoryRB

This object is created from a **SmoothTrajectory** and performs a constrained smoothing of the linear states. The results are stored as a list of non-collapsed problem-specific objects.

4. Problem specific methods

This section describes the functions required to be implemented for each specific problem class. It is broken into subsections detailing the methods needed depending on the choice of estimation algorithm

4.1 Filtering

All methods needed are defined in **ParticleFilteringBase**.

- *sample_input_noise*(self,u) Given the measured input to the system perturb it according to the noise distribution. Preferably leaving any parts with Gaussian noise acting only on linear states intact, and handling that in the update part with a Kalman filter for those states. The output from this function is what gets forwarded to the *update* function.
- *update*(self, data) Update the system state according to the (perturbed) input u. A class, **KalmanFilter** is provided for convenient handling of the conditionally linear Gaussian states.
- *measure*(self, y) Should return the likelihood of the provided measurement conditioned on the particles current state. And for CLG states the estimate should be updated with the new information.

4.2 Smoothing

All the methods from the filtering case are needed and in addition to those specified in **ParticleSmoothingBase**.

- *next_pdf*(self, next_cpart, u) Evaluate the probability density function for the next state conditioned on the current state and the input
- *collapse*(self) Return an (user defined) object containing the nonlinear state and sampled linear Gaussian states.
- *sample_smooth*(self, filt_traj, ind, next_cpart) Return a collapsed particle with the linear states sampled from the distribution obtained by conditioning on the next particle state.

4.3 Rao-Blackwellized Smoothing

All the methods from the smoothing case are needed and in addition those specified in **ParticleSmoothingBaseRB**.

- *clin_update*(self, u) Update the CLG states conditioned on the non-linear states.
- *clin_measure*(self, y) Do a measurement update of the CLG states conditioned on the nonlinear states.

- *clin_smooth*(self, z_next, u) Perform a backward smoothing step for the current CLG states conditioned on the future state.
- *get_nonlin_state*(self) Return an object containing the estimate of the non CLG states.
- *set_nonlin_state*(self, eta) Set the nonlinear state to the data provided.
- *get_lin_est*(self) Return the mean and covariance of the estimate for the CLG states.
- *set_lin_est*(self, lest) Set the mean and covariance of the estimate for the CLG states.
- *linear_input*(self, u) Return the part of the input vector only affecting CLG states.

5. Kalman filtering/smoothing

There are two classes provided for linear Gaussian filtering and smoothing for sampled system, **KalmanFilter** and **KalmanSmoother**. The matrices used can either be dense or sparse of the type defined in **scipy.sparse**.

The **KalmanFilter** class provides methods for creating a filtered estimate provided the A, B, C, D matrices describing the system and the covariance matrices describing the noises affecting the system and measurements.

The provided methods are:

- *time_update* Performs a time update of all the system states according to the specified dynamics using the provided input. Allows for specifying all the relevant matrices to allow for time-varying systems.
- *predict* Does the same calculations as the *time_update* function but without updating the internal object state.
- *meas_update* Updates the current estimate using the provided measurement, allows for specifying the measurement matrix to allow for time-varying dynamics.

The **KalmanSmoother** class provides one additional method,

- *smooth* Makes a smoothing estimate of the current state by incorporating the information from the next state estimate (mean and covariance)

6. Implemented model classes

This chapter details a few models for specific types of problems which are already implemented so they can be reused or expanded for similar problems and serve as additional documentation for how to implement new problem classes.

6.1 Mixed linear/nonlinear Gaussian

Provides implementation of filtering and smoothing for problems of the type

$$\begin{aligned} \zeta_{t+1} &= f_\xi(\xi_t) + A_\xi(\xi_t) z_t + v_{\xi,t} \\ z_{t+1} &= f_z(\xi_t) + A_z(\xi_t) z_t + v_{z,t} \\ y_t &= h(\xi_t) + C(\xi_t) z_t + e_t \end{aligned}$$

where all the noise sources $v_{\xi,t}, v_{z,t}, e_t$ are white Gaussian given the nonlinear estimate ξ_t.

To solve this type of estimation problem it is thus only required to implement the function for evaluating the nonlinear functions f_ξ, f_z and the (time-varying) matrices depending on ξ.

6.2 Differential drive with uniform noise

This a model for the movement in the plane of a robot that can be modelled as using a differential type drive. The inputs are wheel encoder measurements of the wheel positions. Due to limitations of the angular resolution there is a remaining uncertainty of the actual rotational position of the wheel, which in turn yields an uncertainty of the actual robot position. This is modelled as a uniform noise on the wheel encoder measurements.

To model other uncertainty and effects such as wheel slip there is also an additional Gaussian noise affecting the orientation of the robot.

The model as presented here becomes just a dead-reckoning solution, to be actually useful it has to be extended with a measurement model to determine which particles have a high likelihood and which can be discarded.

For one such possible extension see [Berntorp and Nordh, 2013].

$$\xi_{k+1} = f(\xi_k, u_k, v_k, w_k)$$

Here, $\xi_k = (x_k \ y_k \ \theta_k \ P^R_{k-1} \ P^L_{k-1} \ P^R_{k-2} \ P^L_{k-2})^{\mathrm{T}}$ is the state vector, with $P^{R,L}_k$ being the right and left wheel encoder positions at time index k. Further, the input $u_k = (P^R_{k+1} \quad P^L_{k+1})^{\mathrm{T}}$ and the wheel encoder noise vector is assumed uniformly distributed according to $v_k \sim U(-\alpha, \ \alpha)$. The process noise w_k only enters in the θ-state with variance Q_w. After introducing

$$\bar{\theta}_k = \theta_k + \begin{pmatrix} \frac{1}{2l} & -\frac{1}{2l} \end{pmatrix} \begin{pmatrix} P^R_{k-2} \\ P^L_{k-2} \end{pmatrix} + \begin{pmatrix} \frac{1}{2l} & -\frac{1}{2l} \end{pmatrix} (u_k + v_k),$$

the kinematics vector $f(\xi_k, u_k, v_k, w_k)$ is

$$f(\xi_k) = \begin{pmatrix} 1 & 0 & 0 & \frac{1}{4}a & \frac{1}{4}a & -\cos\theta_k & -\cos\theta_k \\ 0 & 1 & 0 & \frac{1}{4}b & \frac{1}{4}b & -\sin\theta_k & -\sin\theta_k \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix} \xi_k + \begin{pmatrix} \frac{1}{4}\cos\bar{\theta}_k & \frac{1}{4}\cos\bar{\theta}_k \\ \frac{1}{4}\sin\bar{\theta}_k & \frac{1}{4}\sin\bar{\theta}_k \\ 0 & 0 \\ 1 & 0 \\ 0 & 1 \\ 0 & 0 \\ 0 & 0 \end{pmatrix} (u_k + v_k) + \begin{pmatrix} 0 \\ 0 \\ \bar{\theta}_k + w_k \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix}.$$

where l is the distance between the center of the wheels, $a = \cos\theta_k - \cos\bar{\theta}_k$, and $b = \sin\theta_k - \sin\bar{\theta}_k$.

7. Example

As an example of the mixed linear/nonlinear model described above we have the system

$$\begin{aligned} x_{k+1} &= \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} x_k + \begin{pmatrix} 0 & 0 \\ 1 & -1 \\ 0 & 0 \end{pmatrix} (u_k + v_k) + w_k \\ y_k &= \begin{pmatrix} x_k(3) & 0 & 0 \end{pmatrix} x_k + e_k \\ v_k &\sim N\left(0, \begin{pmatrix} 0.12 & 0 \\ 0 & 0.12 \end{pmatrix}\right) \\ w_k &\sim N\left(0, \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0.01 \end{pmatrix}\right) \\ e_k &\sim N(0, 1) \end{aligned}$$

This example is implemented in the file `simple_particle.py` and running `simple_test.py` produces the results that can seen in Figure 1.

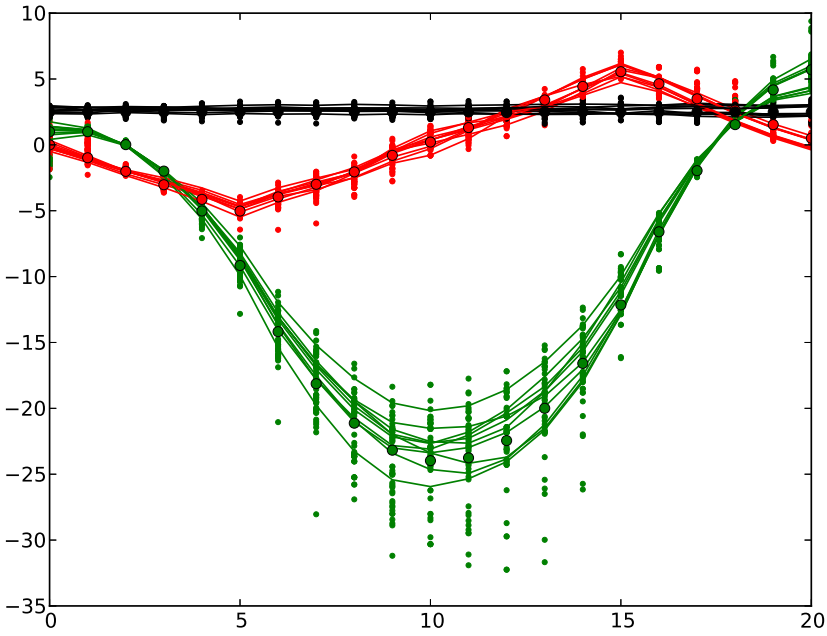

Figure 1 Dots are filtered estimates, lines are smoothed trajectories. The big dots are the true states. x(1) - green, x(2) - red, x(3) - black

8. References

Arulampalam, M., S. Maskell, N. Gordon, and T. Clapp (2002): "A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking." *IEEE Transactions on Signal Processing*, **50:2**, pp. 174–188.

Berntorp, K. and J. Nordh (2013): "Rao-Blackwellized particle smoothing for occupancy-grid based SLAM using low-cost sensors." In *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems*. Tokyo, Japan. *Submitted*.

Doucet, A., S. Godsill, and C. Andrieu (2000): "On sequential Monte Carlo sampling methods for Bayesian filtering." *Statistics and Computing*, **10:3**, pp. 197–208.

Appendix

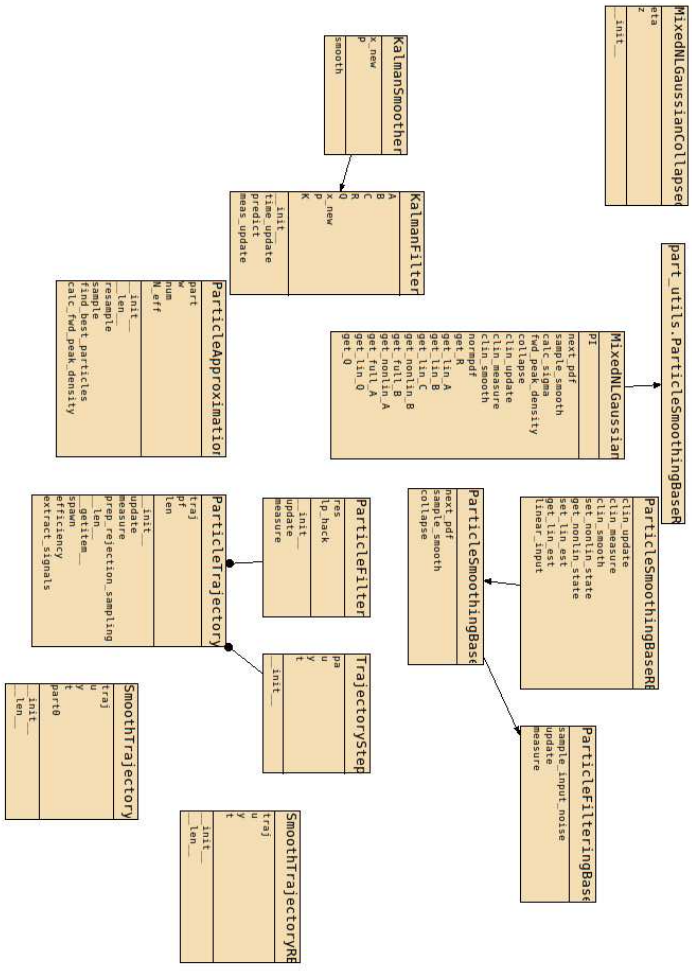

Figure 2 A figure describing all classes and their methods used in the framework.